October 11–13, 2016 Berlin, Germany

OCF for resource-constrained environments

Kishen Maloor, Intel

Outline

Introduction

Brief background in OCF Core

Constrained environment charactertics

IoTivity-Constrained architecture

Building applications

Porting to new environments

Introduction

What is OCF?

- Industry consortium with several member companies
- Publish open IoT standards
- Standards supported by open-source reference code

Introduction

What is this talk about?

- IoTivity-Constrained
  - New open source stack
  - Reference Implementation of OCF standards for resource-constrained devices
  - Easily customizable to any constrained platform configuration

Introduction

RFC 7228: Classes of Constrained Devices

Name	data size (e.g., RAM)	code size (e.g., Flash)
Class 0, C0	<< 10 KiB	<< 100 KiB
Class 1, C1	~ 10 KiB	~ 100 KiB
Class 2, C2	~ 50 KiB	~ 250 KiB

- Challenge: Must accommodate (at a minimum) OS + Network stack + drivers + OCF protocol + OCF application

IoTivity-Constrained vs IoTivity

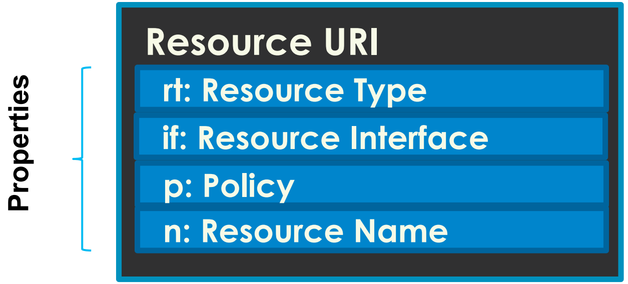
IoTivity-Constrained	IoTivity
For resource-constrained devices	For multi-function devices
Battery-powered door locks, tiny wireless sensors embedded in ceiling etc.	PCs, smartphones, gateways etc
Devices run small OS	Devices run full-featured OS
Zephyr, Contiki, RIOT OS etc.	Linux, Android, Tizen, Windows etc.

Both implementations are protocol compatible

Brief background in OCF Core

OCF resource model

- RESTful design: Things modeled as resources with properties and methods
- CRUDN operations on resources (GET / OBSERVE, POST, PUT, DELETE)
- OCF roles
  - Server role: Exposes hosted resources
  - Client role: Accesses resources on a server



Brief background in OCF Core

OCF Protocols

- Messaging protocol
  - CoAP (RFC 7252)
  - Resource discovery and reliability
- CBOR (RFC 7049) encoding of OCF payloads
- DTLS-based authentication, encryption and access control lists*
- Uses UDP/IP transport; being adapted to Bluetooth

*Refer to the OCF Security spec at https://openconnectivity.org/resources/specifications

Brief background in OCF Core

“Well-known” OCF resources

Functionality	Fixed URI
Discovery	/oic/res
Device	/oic/d
Platform	/oic/p
Security	/oic/sec/*
…	…

Refer to the OCF Core spec at https://openconnectivity.org/resources/specifications

Brief background in OCF Core

OCF request/response examples

Resource discovery

Multicast GET coap://224.0.1.187:5683/oic/res

Unicast response

[URI: /a/light; rt = [“oic.r.light”], if = [“oic.if.rw”],
p= discoverable, observable]

Brief background in OCF Core

OCF request/response examples

GET and PUT requests

Unicast GET coap://192.168.1.1:9000/a/light

Unicast response

[URI: /a/light; state = 0, dim = 0]

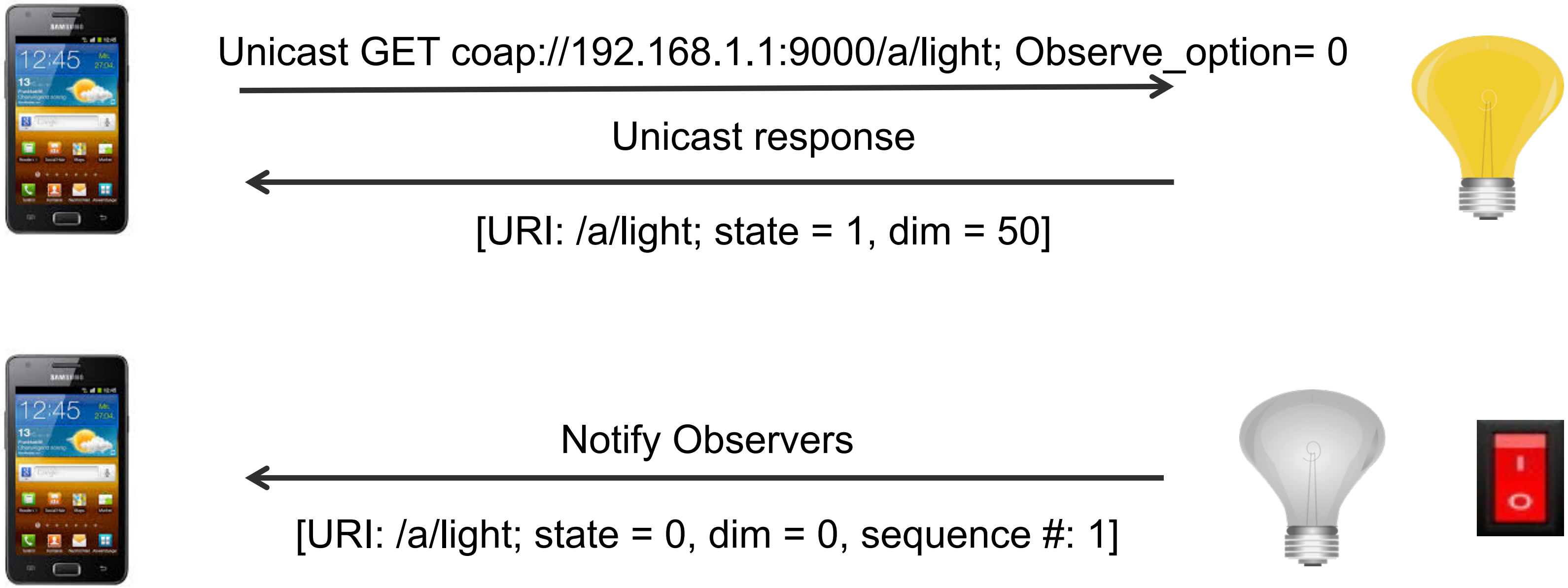
Unicast PUT coap://192.168.1.1:9000/a/light
PayLoad: [state=1;dim=50]

Unicast response

Status = Success

Brief background in OCF Core

OCF request/response examples **OBSERVE / Notify**



Working in constrained environments

Hardware related

- Low RAM and flash capacity
- Low power CPU with low clock cycle
- Execution efficiency
- Allow selection of feature subset to fulfill application’s purpose

Working in constrained environments

Software related

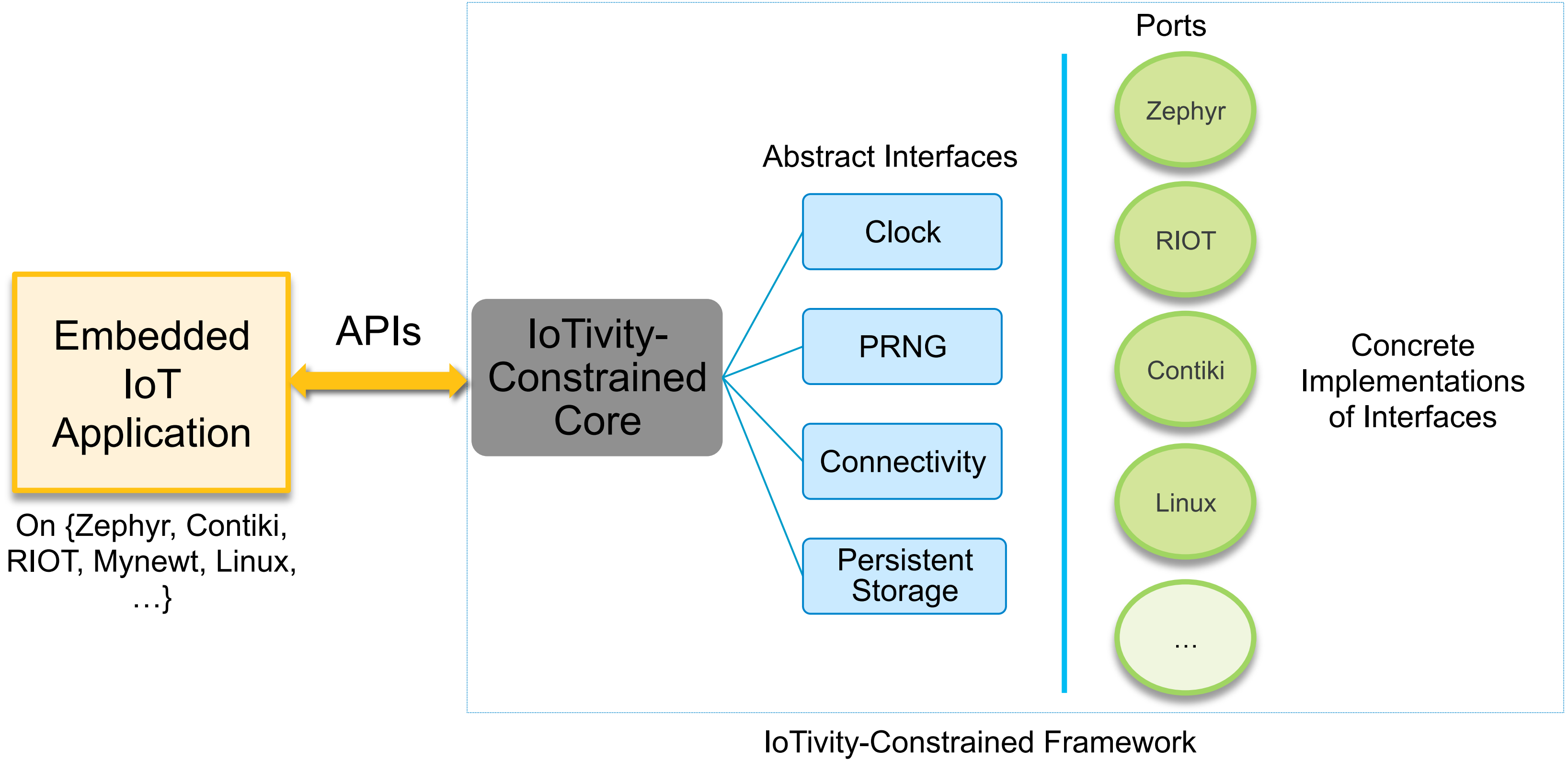
- Lightweight OS
- No dynamic memory management
- Multiplicity of OS, network stack, hardware platform and peripheral options
- Varying execution context design and scheduling strategy

IoTivity-Constrained architectural goals

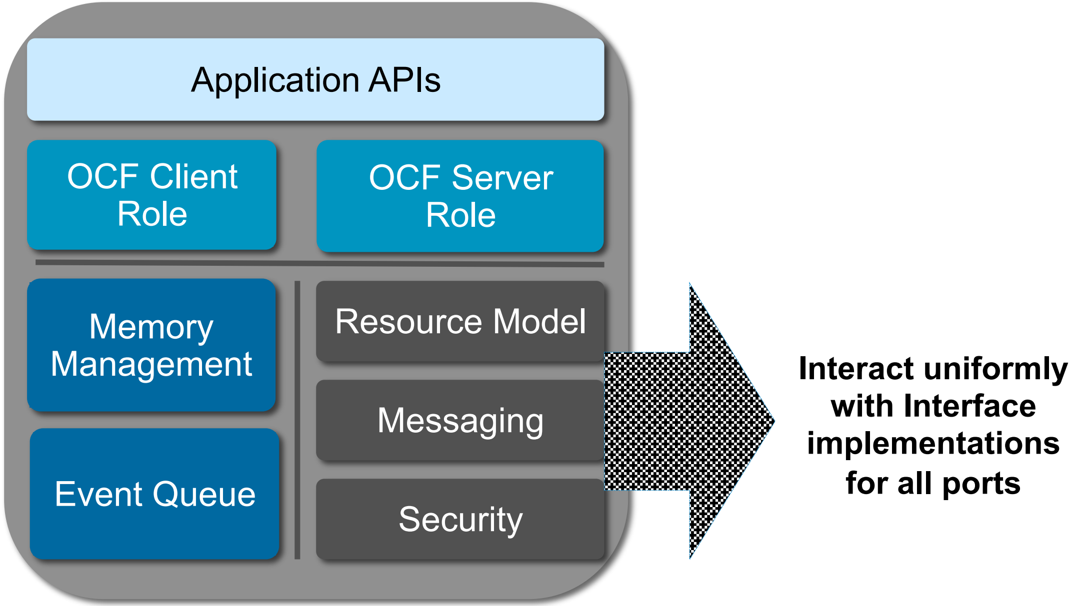
At a high level

- Cross-platform core
- Abstract interfaces to system clock, connectivity, PRNG, persistent storage
- Rapid porting to new environments
- Static memory allocation
- Modular design

IoTivity-Constrained architecture

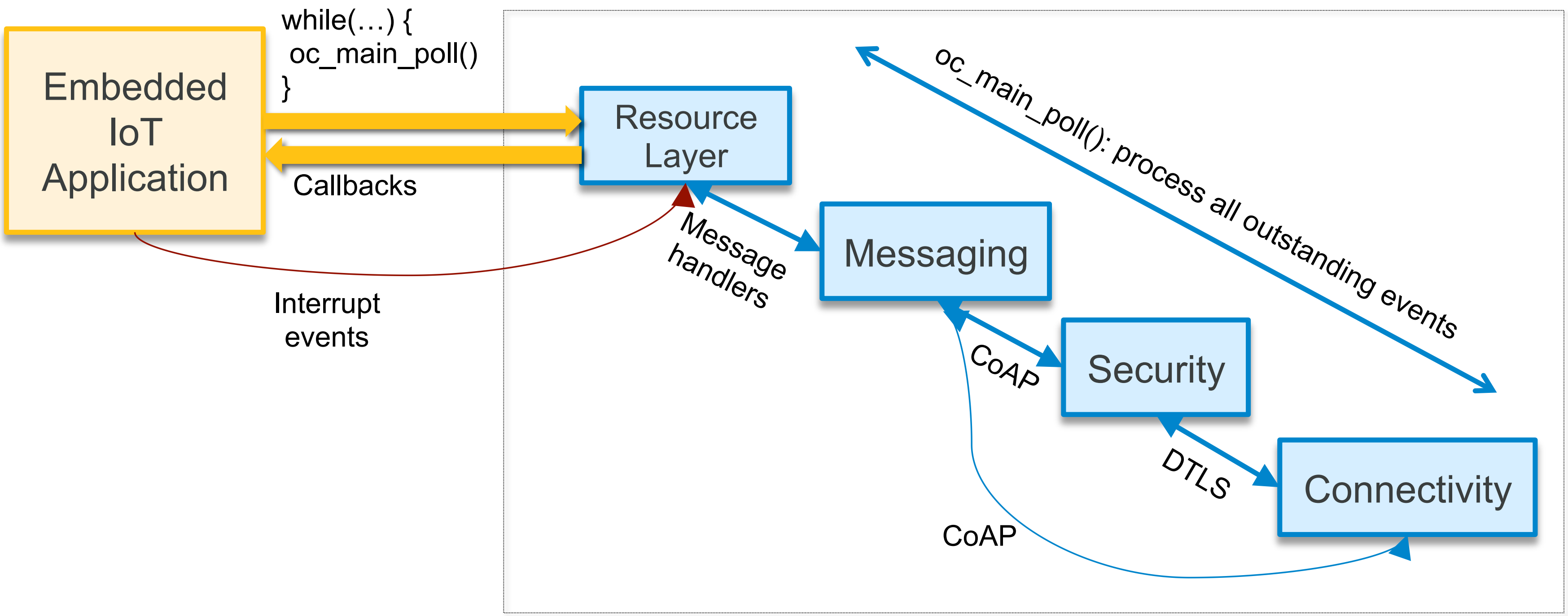


IoTivity-Constrained core



IoTivity-Constrained event loop

Event loop execution



IoTivity-Constrained event loop

Enter tickless idle mode during periods of inactivity

```
...
// Initialize a semaphore
while (1) {
    oc_clock_time_t next_event = oc_main_poll();
    // next_event is the absolute time of the next scheduled event in clock ticks
    // Meanwhile, do other tasks or sleep (e.g., wait on semaphore)
...
// Framework invokes a callback when there is new work
static void signal_event_loop(void) {
    // Wake up the event loop (e.g., signal the semaphore)
}
```

Building applications

Application structure

- Incorporate OCF’s client or server roles or both
- Implement a set of callbacks
  - Initialization (Client / Server)
  - Defining and registering OCF resources (Server)
  - Resource handlers for all supported methods (Server)
  - Response handlers for all requests (Client)
  - Entry point for issuing requests (Client)
- Framework configuration at build-time (config.h)

Building applications

Setting the callbacks

Main task in application

```
main() {
  static const oc_handler_t handler = {.init = app_init,
                                        .signal_event_loop = signal_event_loop,
                                        .register_resources = register_resources };
...
  oc_main_init(&handler);
...
  while (1) {
    oc_clock_time_t next_event = oc_main_poll();
...
```

Building applications

Initialization **Client / Server**

```
void app_init(void) {
  oc_init_platform("Intel", NULL, NULL);
  oc_add_device("/oic/d", "oic.d.light", "Lamp", "1.0", "1.0", NULL, NULL);
  oc_storage_config(“./creds”);
}
```

- Populate standard OCF resources (platform / device)
- oc_storage_config is defined in the implementation of the storage interface for a target

Building applications

Defining a resource

Server-side

```
....
void register_resources(void) {
  oc_resource_t *res = oc_new_resource("/a/light", 1, 0);
  oc_resource_bind_resource_type(res, "core.light");
  oc_resource_bind_resource_interface(res, OC_IF_R);
  oc_resource_set_default_interface(res, OC_IF_R);
  oc_resource_set_discoverable(res, true);
  oc_resource_set_observable(res, true);
  oc_resource_set_request_handler(res, OC_GET, get_light, NULL);
  oc_add_resource(res);
}
```

Building applications

Defining a resource handler

Server-side

```
....
bool light_state;
int brightness;
....
static void get_light(oc_request_t *request, oc_interface_mask_t interface, ...) {
  oc_rep_start_root_object(); // Call oc_get_query_value() to access any uri-query
  oc_rep_set_boolean(root, state, light_state);
  oc_rep_set_int(root, brightness_level, brightness);
  oc_rep_end_root_object();
  oc_send_response(request, OC_STATUS_OK);
}
```

Building applications

Resource discovery

Client-side

```
oc_do_ip_discovery("oic.r.light", &discovery, NULL);
....
oc_server_handle_t light_server;
char light_uri[64];
...
oc_discovery_flags_t discovery(..., const char *uri, ..., oc_server_handle_t *server,
                                ,...) {
  strncpy(light_uri, uri, strlen(uri));
  memcpy(&light_server, server, sizeof(oc_server_handle_t));
  return OC_STOP_DISCOVERY; // return OC_CONTINUE_DISCOVERY to review other resources
}
```

Building applications

Issuing a request

Client-side

```
oc_server_handle_t light_server; // Populated in the discovery callback
char light_uri[64];
...
oc_do_get(light_uri, &light_server, "unit=cd", &get_light, LOW_QOS, NULL);
...
```

Building applications

Handling a response **Client-side**

```
void get_light(oc_client_response_t *data) {
  oc_rep_t *rep = data->payload;
  while (rep != NULL) { // rep->name contains the key of the key-value pair
    switch (rep->type) {
    case BOOL:
      light_state = rep->value_boolean; break;
    case INT:
      brightness = rep->value_int; break;
    }
    rep = rep->next;
  }
}
```

Building applications

Separate response: For slow resources

Server-side

```
oc_separate_response_t temp_response; // Separate response handle for a request
...
static void get_temp(oc_request_t *request, oc_interface_mask_t interface) {
  oc_indicate_separate_response(request, &temp_response);
}
...
void response_ready(…) {
...
  oc_set_separate_response_buffer(&temp_response);
  oc_rep_start_root_object();
  oc_rep_set_int(root, temp, temperature);
  oc_rep_end_root_object();
  oc_send_separate_response(&temp_response, OC_STATUS_OK);
```

Building applications

Scheduling events

Client / Server

- Trigger callback after a period of time

```
...
oc_event_callback_retval_t run_after_a_sec(void *user_data)
{
...
  return DONE; // Tears down the callback
  // return CONTINUE instead to be called back once more after the preset interval
}
...
oc_set_delayed_callback(NULL, &run_after_a_sec, 1);
...
```

- May be used for scheduling one-off or periodic request issuances

Building applications

Handling Interrupts

Server-side

- Applications can define and signal a callback from an external context (eg. ISR)
- Callback executed on task that runs the event loop

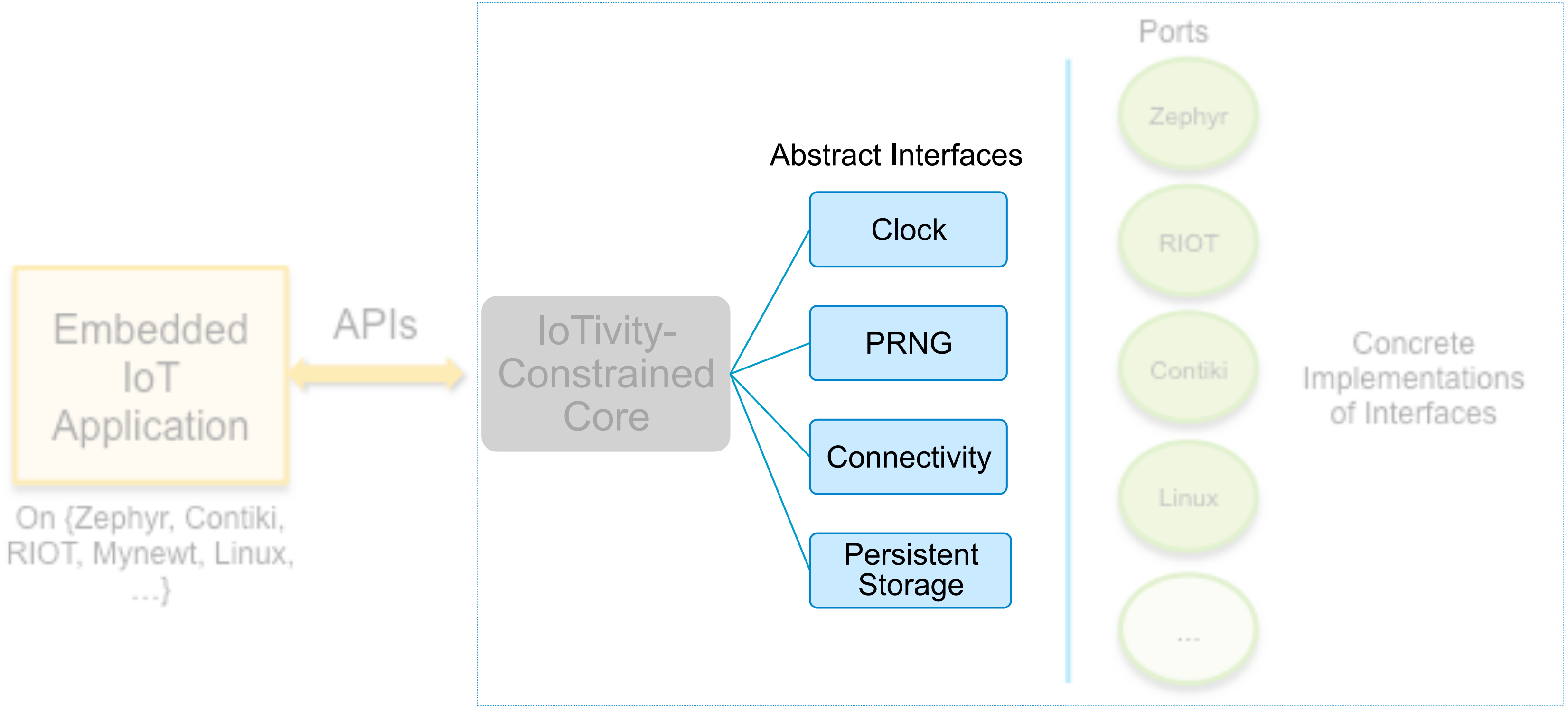
```
oc_define_interrupt_handler(temp_sensor) { // Callback definition
...
}
...
static void app_init(void){
 oc_activate_interrupt_handler(temp_sensor); // Callback registration
...
void temp_sensor_isr() {
...
 oc_signal_interrupt_handler(temp_sensor); // Callback Signaling
```

Building applications

Framework configuration

- Set at build-time in a file config.h
  - Number of application resources
  - Number of request/response buffers
  - Maximum payload sizes
  - Memory pool sizes
  - Max # of DTLS peers
  - DTLS connection timeout
  - …

Porting to new environments



IoTivity-Constrained Framework

Porting to new environments

Clock interface

```
// Set clock resolution in IoTivity-Constrained's configuration file: config.h
#define OC_CLOCK_CONF_TICKS_PER_SECOND (...)
typedef uint64_t oc_clock_time_t; // timestamp field width

// Declared in port/oc_clock.h
// Implement the following functions using the platform/OS's APIs, For eg. on Linux
// using clock_gettime()
void oc_clock_init(void);
oc_clock_time_t oc_clock_time(void);
unsigned long oc_clock_seconds(void);
void oc_clock_wait(oc_clock_time_t t);
```

Porting to new environments

Connectivity interface

```
// Declared in port/oc_connectivity.h
// Implement the following functions using the platform's network stack.
int oc_connectivity_init(void);
void oc_connectivity_shutdown(void);
void oc_send_buffer(oc_message_t *message);
void oc_send_multicast_message(oc_message_t *message);
uint16_t oc_connectivity_get_dtls_port(void);
```

- oc_message_t contains remote endpoint information (IP/Bluetooth address), and a data buffer

Porting to new environments

Connectivity interface: network event synchronization

- Capture incoming messages by polling or with blocking wait in a separate context and construct an oc_message_t object
- Message injected into framework for processing via oc_network_event() call
- Based on nature of OS or implementation, might require synchronization

```
void oc_network_event_handler_mutex_init(void);
void oc_network_event_handler_mutex_lock(void);
void oc_network_event_handler_mutex_unlock(void);
```

Porting to new environments

PRNG interface

```
// Declared in port/oc_random.h
// Implement the following functions to interact with the platform's PRNG
void oc_random_init(void);
unsigned int oc_random_value(void);
void oc_random_destroy(void);
```

Porting to new environments

Persistent storage interface

```
// Declared in port/oc_storage.h
// Implement the following functions to interact with the platform's persistent storage
// oc_storage_read/write must implement access to a key-value store
int oc_storage_config(const char *store_ref);
long oc_storage_read(const char *key, uint8_t *buf, size_t size);
long oc_storage_write(const char *key, uint8_t *buf, size_t size);
```

Conclusion

Pointers to code

- Source code: https://gerrit.iotivity.org/gerrit/gitweb?p=iotivity-constrained.git
- IoTivity mailing list: iotivity-dev@lists.iotivity.org

- Project actively developed
  - Ports for RIOT OS, Zephyr, Contiki and Linux
  - Feature gaps, enhancements, pass OCF certification tests
  - Project, code and API documentation
- Your involvement and contributions are welcome!

Q & A